TetStreamer: Compressed Back-to-Front Transmission of Delaunay Tetrahedra Meshes

Urs Bischoff and **Jarek Rossignac**
Georgia Institute of Technology, 85 Fifth Street NW, Atlanta, GA 30332-0760
Phone 404-894-0671, Fax: 404-894-0673, Email: jarek@cc.gatech.edu

Abstract

We use the shorts tet and tri for tetrahedron and triangle. TetStreamer encodes a Delaunay tet mesh in a back-to-front visibility order and streams it from a server to a client (volumetric visualizer). During decompression, the server performs the view-dependent back-to-front sorting of the tets by identifying and deactivating one free tet at a time. A tet is free when all its back faces are on the sheet. The sheet is a tri mesh separating active and inactive tets. It is initialized with the back-facing boundary of the mesh. It is compressed using EdgeBreaker and transmitted first. It is maintained by both the server and the client and advanced towards the viewer passing one free tet at a time. The client receives a compressed bit stream indicating where to attach free tets to the sheet. It renders each free tet and updates the sheet by either flipping a concave edge, removing a concave valence-3 vertex, or inserting a new vertex to split a tri. TetStreamer compresses the connectivity of the whole tet mesh to an average of about 1.7 bits per tet. The footprint (in-core memory required by the client) needs only to hold the evolving sheet, which is a small fraction of the storage that would be required by the entire tet-mesh. Hence, TetStreamer permits to receive, decompress, and visualize or process very large meshes on clients with a small in-core memory. Furthermore, it permits to use volumetric visualization techniques, which require that the mesh be processed in view-dependent back-to-front order, at no extra memory, performance or transmission cost.

1. Introduction and prior art

In this paper, we use the abbreviations “tet” for “tetrahedron” and “tri” for “triangle”.

Direct volume rendering [1] is used in medical and other scientific domains to visualize scalar fields defined in terms of a tet interpolation of scalar values at irregularly distributed 3D samples. The lighting model assumes that the volume is filled with a translucent material that emits its own light, absorbs incoming light, and partially transmits light coming from the back. The coefficients that govern this behavior at any point in space are functions of the value of the scalar field at that point. Proper rendering requires visiting the tets in visibility order (back-to-front) along each ray coming to the eye through a pixel. We propose a new approach, called *TetStreamer*, for the transmission of tet meshes between a **server** and a **visualizer**—a remote client or a local visualization board. Our approach addresses three issues simultaneously: Footprint, Order, and Compression. We discuss them below.

Footprint: Typical data structures for tet meshes use more between 100 and 300 bytes per node. Hence, the client—a visualization accelerator board or a mobile device—may not have sufficient on board memory to store the entire tet mesh. Since its job is only to visualize the mesh, we can use a streaming solution where the client decodes the tets one at a time, renders them, and then discards their representation and their vertices, when these are no longer needed, because all their incident tets have been processed. In our

approach, the client or visualization unit needs only maintain a **sheet** of triangles that separate already rendered tets from the ones that have not yet been received. A similar concept has been exploited previously to reduce the footprint required by a client for rendering a tri mesh by buffering on the client only the vertices on the boundary of previously rendered triangles. This approach either required vertex retransmission, due to hardware limits imposed on the size of the buffer [2] or reorders the tris to reduce the maximum size of the buffer without vertex retransmission [3, 4]. We extend these footprint-reducing ideas to tet meshes, but with the additional constraint of back-to-front transmission order, as discussed below.

Order: The correct volume visualization of tet meshes [1, 5-7] requires that tets be processed in back-to-front visibility order. Several approaches have been proposed to sort tets in back-to-front order [8, 9]. We propose a solution where the tets are visited in back-to-front visibility order by sweeping the sheet forward through the tet mesh and ensuring that it preserves visibility order. Our solution does not require any sorting or preprocessing.

Compression: Previously proposed tet mesh compression approaches have focused on the reduction of the storage necessary for encoding the connectivity of a tet mesh, which usually dominates storage requirements. Two approaches [10, 11] transmit the nodes in the order in which they are encountered by a spiraling traversal of the tet mesh. Reported results encode the connectivity down to less than 2.4 bits using entropy codes [11]. Compressed Encodings of Progressive Tet Meshes [12] use batches of upgrades, which identify a subset of the internal vertices of the mesh and split them, reversing the effects of edge-collapse simplification steps. Once amortized, the connectivity transmission cost is about 5 bits per tet. In contrast to these two families of approaches, we encode the mesh as the evolution of a sheet that sweeps past one tet at a time. Our solution, which in addition to encoding the connectivity of the mesh is transmitting the tets in back-to-front order, requires about **1.7 bits per tet**.

The proposed *TetStreamer* approach offers a significant advantage over previous solutions when all three factors (footprint, order, and compression) are important. Furthermore, due to its simplicity and effectiveness, it may be more broadly valuable for visibility sorting alone, for compression alone, or for the out-of-core processing of tet meshes that have a Delaunay property.

The inspiration of our work comes from the Gatun system [13]. Gatun provides an elegant solution to both the compression and footprint problems for the transmission of tet meshes, although it does not provide the option to transmit the tets in visibility order. To clarify the similarities and differences between the two solutions, we describe the connectivity encoding technique of Gatun using the terminology with which we describe TetStreamer. The Gatun server first encodes and transmits the outer surface of the tet mesh as a compressed tri mesh. We will call it the sheet to establish an analogy with our own terminology. Then, the triangular faces of the sheet are repeatedly visited in the same order by the server and client. Both push the sheet inwards, past one tet at a time, hence peeling the outer layers of the tet mesh. In our terms, initially all tets are **active**. When a tet is passed by the sheet, it becomes **inactive**. Let F be the sheet-face currently visited on the sheet. Let V be the tip vertex of the active tet having F as **base**. A single bit may be used to indicate whether V is on the sheet or not. If V is not on the sheet, it is transmitted and assigned the first available vertex ID on the viewer. When V is already on the sheet, its ID (integer index identifying the vertex in the table of all current vertices

of the sheet) could be encoded using $\log_2(N)$ bits, where N is the current number of vertices on the sheet. To reduce the expected cost of this non-constant term, Gatun renumbers the vertices of the sheet around the base face F. It deviates from the simpler spiraling renumbering scheme, used in the Cut-Border Machine [10, 11], by allocating the lowest indices to the tips of faces that share an edge with F. Then Gatun goes around the three vertices of F and assigns subsequent integers to the other neighboring vertices. Most often, V is the tip of a neighboring face of the sheet and hence is identified by a small integer (0, 1, or 2). In the rare cases where it is not, the global ID of V is encoded using $\log_2(N)$ bits. As tet T is deactivated, the sheet is updated on the client. Note that the update operation in Gatun may require a clean-up phase to remove pairs of faces that are not bounding any active tet. Such removals may change the topology of the sheet by creating handles or separating components. Furthermore, the deactivation of a tet may create non-manifold singularities in the sheet, which complicate the encoding and traversal and which require a more elaborate data structure for representing the sheet on the client. Gatun uses four operators to encode the connectivity. EXPLICIT implies that V is a new vertex. INDEX implies that V is not in the neighborhood of F and is encoded using an absolute ID. FACE implies that V is the tip of a face sharing an edge with F. VERTEX includes all other situations, where V is connected by an edge to a vertex of F. The footprint on the client is reduced to the storage of the sheet and of a small subset of tets. Vertices that are no longer on the sheet are deleted and their ID reclaimed for subsequent vertices. Tets that are no longer needed are identified through a custom garbage collection process and deleted. Note that Gatun does not transmit the tets in back-to-front order. Instead, it uses a ray-casting volume visualization process, exploiting the adjacency information derived from the sheet to track and quickly identify which rays intersect the new tet. The approach is based on the observation that when a ray exits a tet through a face F, it enters the other tet incident upon F. On average Gatun achieve a connectivity encoding of 2.3 bits per tet and a decompression performance of 162K tets per second. It is not restricted to Delaunay meshes. *TetStreamer* builds upon Gatun's idea of encoding how to sweep a sheet, but instead of sweeping the sheet inwards, it starts with the back facing boundary of the tet mesh and sweeps forward (as illustrated in Figure 1). The original motivation for this difference is of course to transmit the tets in back-to-front order. Surprisingly, this additional constraint leads to a simpler algorithm and to significantly better compression when the mesh is a Delaunay tetrahedralization or when it is convex and has no visibility locks. In particular, *TetStreamer* exploits the fact that the sheet never changes topology and remains free from non-manifold situations. This property follows directly from the observation that all the triangles in the sheet are front-facing, hence the sheet remains a depth field. Furthermore, the sheet updates in TetStreamer are reduced to only two trivial operations. The I operation (which corresponds to the EXPLICIT operation in Gatun) is a vertex insertion that splits the base triangle into 3. The F operation (which corresponds to the FACE operation in Gatun) flips an edge and, if the flip has produced a valence-2 vertex, performs an edge-collapse to remove it. Note that INDEX and VERTEX operations needed in Gatun are impossible and hence need not be encoded nor implemented in TetStreamer. Finally, there is no need for garbage collection, since the vertices of the sheet are eliminated only when their valence reaches 2 and since tets are rendered as they are passed through by the sheet and never stored by the client. As a consequence, both the compression and the decompression may be implemented using a simple Corner Table data structures. The server stores the tet connectivity of the tet mesh as an array of integer pairs, 4 pairs per

tet. The client stores the sheet connectivity, using the Corner Table [14], as an array of integer pairs, 3 pairs or each tri present in the sheet. The I, F, and edge-collapse operations may be trivially implemented using local low-level operators.

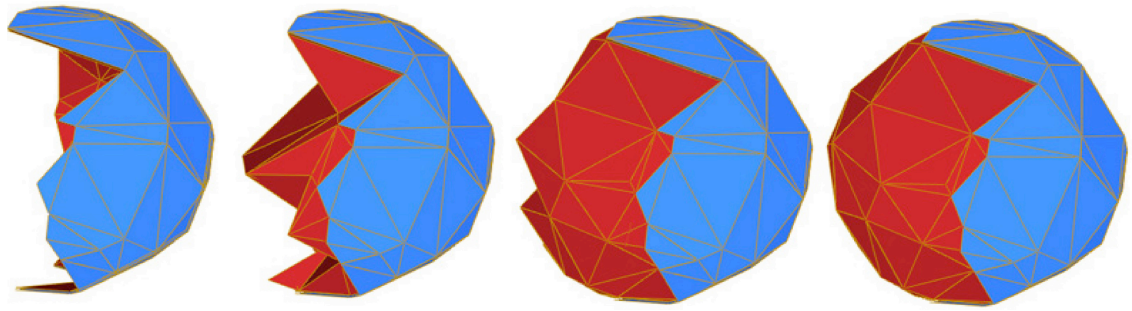


Figure 1*: The viewpoint is on the left. From left to right, the images illustrate the consecutive stages recovered from a back-to-front transmission of the tets of a convex mesh. The compressed stream indicates where and how to attach new tets to the red sheet, which is advanced to pass them, one tet at a time. The connectivity information is compressed to 1.7 bits per tet. The decoder needs only to maintain a tri-mesh representation of the red sheet, which starts as the back facing boundary (left) and ends as the front-facing boundary (right).*

Although our initial implementation of TetStreamer was capable of transmitting more general tet meshes, the version presented here stemmed from the observation that the compression and decompression algorithms can be simplified and the compression improved if we assume that the mesh has at most a depth complexity of one (no overhangs). Furthermore, we assume that a back-to-front visibility order exists (i.e., the mesh is free from occlusion cycles [8]). Note that both conditions hold for meshes that are Delaunay Tetrahedralizations [18]. Therefore, throughout this paper, we assume that the mesh has these properties.

When the tet mesh is not convex, the unfilled portions of its convex hull may be padded with dummy tets [8]. If the mesh is not a Delaunay tetrahedralization, infrequent occlusion locks may occur. They must be detected and resolved [8] by splitting some of the tets.

2. The TetStreamer algorithm

In this section, we explain the TetStreamer compression and decompression algorithms.

The server has access to the representation of the full tet mesh and knows the location of the viewpoint. It first extracts, compresses, and transmits to the client the tri mesh forming the back-facing part of the boundary of the tet mesh. This tri-mesh is used by the client as the initial state of the sheet.

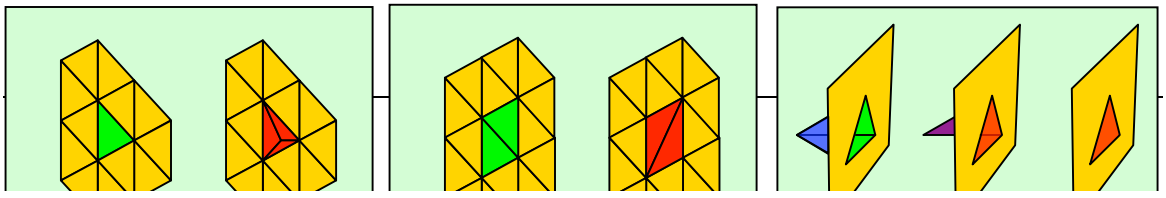
All tets in the mesh are initially marked as active by the server. A tet can be deactivated only when all its back faces are triangles of the sheet. This condition guarantees that visibility order is preserved. Deactivating a tet on the servers amounts to simply changing its label from active to inactive. Processing such a change on the client involves four simple steps: (1) Identify the sheet tris that coincide with the back faces of the tet T, (2) when T has a single back face on the sheet, decode its tip vertex, (3) send the four vertices of T to the graphic sub-system, and (4) update the sheet by replacing the back face(s) of T with its front face(s). We distinguish 3 situations and discuss how we process them on the client.

I-operation: When the tet T has exactly one back face (the base) on the sheet, the client needs to decode and store its tip vertex V. We store all vertex data (location and scalar attributes) in an array large enough to contain the largest sheet. We also maintain, encoded in this array, a stack of unused vertex slots, which have been released by a D-operation (see below) and not yet refilled. An I operation is associated with a face F of the sheet. F is the base triangle of tet T. During decompression, we decode the fourth vertex V of T and store it in the vertex table at the slot referenced by the top entry of the stack of free slots. We pop the stack. Consequently, the new vertex is associated with the ID of that slot. Now, we have the IDs of the four vertices of T and can send them to the graphic subsystem. To reflect the fact that T is deactivated, we update the sheet, replacing F by three new triangles, each one joining an edge of F with V. Figure 2a illustrates this step. One of these three new triangles will take the place of F in the data structure representing the sheet. The other two will be written in the first two empty slots in the Corner Table used to represent the sheet. Hence, we also maintain a stack of IDs of empty triangle slots that are created by a D-operation and consumed by an I-operation. We assume that the base face of the I-operation has been identified prior to the operation. The encoding of that identifier in the compressed bit stream is discussed further in the paper.

F-operation: Consider that the tet T has exactly two back faces that coincide with two tris of the sheet: F_1 and F_2. All the vertices of T are already available to the client. Hence, we can send them to the graphics or volumetric visualization subsystem and need not decode any vertex data nor alter the list of vertices. Let E be the edge common to F_1 and F_2. To update the mesh, we must simply flip E, as shown in Figure 2(b), replacing the back faces of T by its front faces. Note that this operation removes two triangles and replaces them with two other triangles. It does not change the triangle count and does not access the stacks of empty slots. Note that we can identify F_1 and F_2 by their common edge E, which must be concave.

D-operation: When the tet T has three of its back faces, F_1, F_2, and F_3, on the sheet, all its vertices are already available to the client. Hence, we can send them to the volumetric visualization subsystem and need not decode any vertex data. To reflect that T is deactivated, we must replace its 3 back faces by its front face F. As shown in Figure 2(c), this amounts to the decimation of the valence-3 vertex V that is common to the 3 back faces of F. Vertex decimation eliminates one vertex (freeing a vertex slot and pushing it on top of the stack of empty vertex slots) and two triangles (which are also pushed on top of the stack of free triangle slots). Note that the decimation of a valence-3 vertex is equivalent to the flip of one of its three incident edges followed by the collapse of another one of its incident edges, pulling V, which now has valence 2, to one of its two remaining neighbors. The D-operation may be identified by the vertex V, which must be concave and valence-3. Alternatively, it may be encoded as the flip of any one of the 3 edges incident upon V. Because V has valence 3, the client knows that this is not an ordinary flip and may either perform the D-operation as a single step or may first perform the flip and then realize that V has valence 2 and collapse V to one of its neighbors.

In the next section, we explain how TetStreamer identifies the I-triangles that form the base of an I-operation and the F-edges that are flipped to perform an F or D operation.



(a) **(b)** **(c)**

FIGURE 2*: In an I-operation (a), the green face of the mesh (a left) is replace by 3 new red faces (a right). This operation corresponds to the deactivation of a tet that has the green tri as its only back face and has the three red tris as front faces. In an F-operation (b), the two green faces of the mesh (b left) are replaced by the two red faces (b right). This edge-flip operation corresponds to the deactivation of a tet that has the two green tris as back faces and has the two red tris as front faces. Note that the edge between the two red tris cannot be concave. In the D operation (c), the viewpoint is on the right. The three triangles (two appear green and one is semi-transparent blue) are the 3 back faces of a tetrahedron T (c left). They will be replaced by a single red triangle (c right) that is the front face of T. The replacement is a D-operation which first flips the bottom edge of the blue triangle (c center), creating a valence-2 vertex V (the tip of the violet triangle, which appears twice in the mesh). Then, one of the edges of the violet triangle is collapsed, bringing V to one of the vertices of the red triangle.*

3. The TetStreamer streams of I and F bits

Since visibility imposes only a partial order on the tets, the server has some freedom in selecting which of the free tets is the next one to be deactivated. Because we do not want to encode any tri or edge ID explicitly as in [13], we use a batch approach similar to the one proposed in [15]. Both the server and the client perform the same traversal of the sheet. In this process, they visit the faces and edges of the sheet one by one and decide whether a particular tri is to be split by an I-operation or whether a particular edge is to be flipped by an F-operation (which may automatically trigger an edge collapse to finish a D-operation as explained above).

The compression repeats a sequence of two-phases, the I-phase and the F-phase, until all tets are transmitted.

The I-phase performs only I-operations. It visits all tris of the sheet. If the current tri F is marked as **dead**, the I-phase does nothing. Otherwise, it reads the next bit from the decompressed I-stream. If the bit is 0, then F is marked as dead and will never be considered again for an I-operation. If the bit is 1, an I-operation is performed. It replaces F with three new triangles that are as not dead. They are immediately considered as candidates for an I-operation recursively. The depth of recursion rarely exceeds 2. Each time a new triangle is created by an I-operation, its bounding edges are marked as **unlocked**. This marking will be exploited by the subsequent F-phases. At the end of an I-phase, no I-operation is possible and all tris are dead.

The F-phase performs only F-operations, possibly followed by an edge collapse to yield a D-operation. It visits only the **concave** edges of the sheet. Edges are marked as either **locked** or not. When an unlocked concave edge E is encountered, the client reads the next bit in the F-stream. If the bit is 0, then E is locked and will remain so until one of the two triangles incident upon E is removed by an edge flip, an edge collapse, or a triangle split. If the bit is 1, E is flipped. If, before the flip, the edge was incident upon a concave valence 3 vertex, the flip is followed by an edge collapse, as explained above to

implement the D-operation. Each time a new triangle is created by an F-operation, its bounding edges are unlocked. Each time an edge is collapsed, the edge where the two triangles removed by the collapse were attached is also unlocked. Finally, all new triangles created during the F-phase are marked as not dead. The F-phase terminates when all edges are locked or convex.

4. Compression and expected entropy of the F and I streams

The I-stream is a sequence of I-symbols. Similarly, the F-stream is a sequence of F-symbols. Each F-symbol and each I-symbol is either a 1-symbols and 0-symbols. The I- and F-streams have different frequencies of 1-symbols, which we exploit to compress these streams with Entropy codes. We explain these biases here and confirm them on experimental results below.

Let us first discuss the entropy of the **I-stream**. In a typical mesh of T tets, the number of boundary faces is negligible in front of the number of internal faces. The total number F of faces is roughly 2T, since each internal face is used by 2 tets and each tet uses 4 faces. Therefore, a trivial code could be used: "1" for the 1-symbol and "0" for the 0-symbol. It requires a bit per face and hence 1 bit per tet. However an entropy code may easily compress the I-stream further. First note that the number of 1-symbols equals V minus the number of vertices on the back face, since each 1-symbols corresponds to the creation of a new vertex. Also note that the number of 0-symbols is usually less than F–V, which is roughly 2T–V. The actual number of 0-symbols is usually even smaller, since some internal faces do not require receiving an I-symbol because they are removed by the F-phase. In most meshes, the number of tets varies between 4.5V and 5.5V, depending on the sampling and algorithm used to build the tet mesh. For simplicity, assume that T=5V. Hence, the ratio r_I of the number of 1-symbols to the number of 0-symbols is 1-to-9, leading to an entropy of 0.47 bits per face, or 0.95 bit per tet.

Let us now turn to the entropy of the **F-stream**. Convex edges cannot be flipped. Hence *TetStreamer* encodes an F-symbols for each concave edge configuration it encounters. A configuration is defined by the two incident triangles. In general, an internal edge E has an average of 6 incident tets. When E is first created, it is convex, so no symbol needs to be transmitted to indicate that E cannot be flipped. Then one of its incident triangles is replaced, either through an edge flip or an I-operation. Usually, E is still convex. Let us say that in 50% of the cases, a further replacement of one of the incident triangles make E concave. The edge usually survives a fourth and fifth evolution of its configuration before being flipped. Hence, we expect to see in the F-stream one 1-symbol and an average of 1.5 0-symbols per edge. As a consequence, the entropy of the F-stream is close to 2.4 bit per edge. There are on average 6 tets incident upon an internal edge and there are 6 edges per tet. So, the expected cost of the F stream is about a 2.4 bits per tet.

When a concave, valence-3 vertex is first created, we pick one of its incident edges and decode its F-symbol. Because each vertex (except for the external vertices on the visible portion of the boundary of the tet mesh) must be deleted by a D-operation, there are roughly V cases where that F-symbol is a 1-symbol. If we read this F-symbol before we read the I-symbol of the only new triangle incident upon the valence 3 vertex, we save one I-symbol and two F-symbols for the other two concave edges incident upon the valence-3 vertex. This amounts to a total saving of V 0-symbols in the I-stream and of 2V

1-symbols in the F stream. This represents an economy of about 0.1 bit per tet for the I-stream and of 0.4 bit per tet for the F-stream.

Hence, the total expected entropy cost is 2.0 bits per tet.

5. Experimental Results

The right column in TABLE 1 shows the benefit of this prediction scheme, which reduces the average entropy to less than 1.69 bits per tet. These results compare favorably to those reported in [11, 13, 17].

	#vertices	#tets	#back faces	#front faces	#internal faces	# I_0	# I_1	# F_0	# F_1	E_I	E_F	E/tet
tet100	100	505	39	23	979	664	73	221	432	0.47	0.92	1.87
tet120	120	382	115	89	662	493	49	113	333	0.44	0.82	1.57
tet2286	2286	13454	565	505	26373	23184	1975	3947	11479	0.40	0.82	1.68
tet3000	3000	19189	257	251	38124	29948	2853	5134	16336	0.43	0.79	1.62

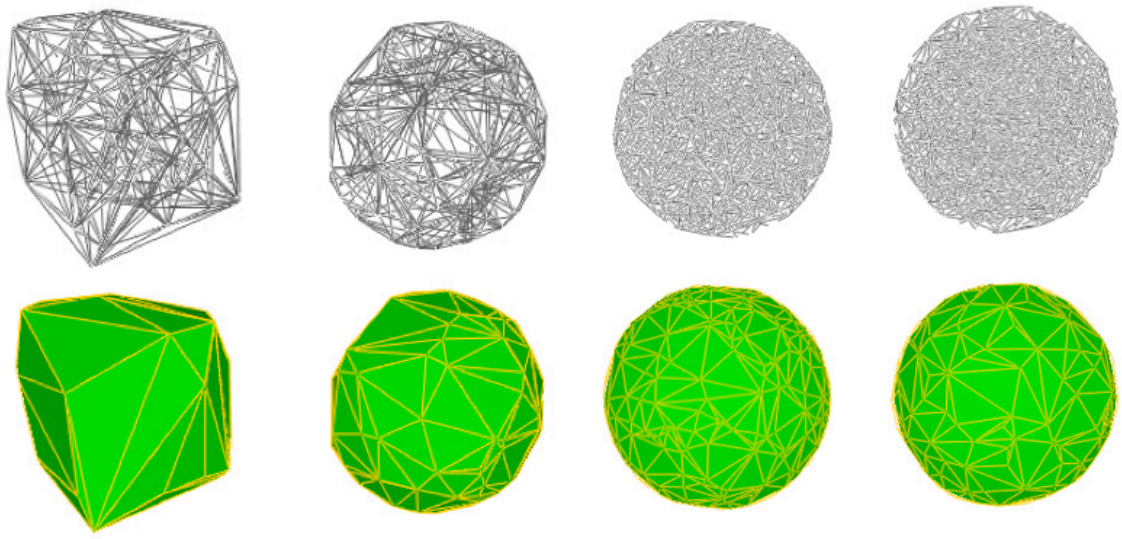
***TABLE 1**: From left to right, the columns contain: the name of the model, the number of vertices, the number of tets, the number of external faces, the number of internal faces, the number of I0, I1, F0, F1 symbol transmitted, the entropy per symbol of the I stream and of the F stream, and finally the total entropy per tet.*

TABLE 2 shows the cost and statistics for the initial states of the corresponding sheets.

	#back faces	# bits to encode back faces	# encoded bits per tet
tet100	39	89	0.18
tet120	115	253	0.66
tet2286	565	1183	0.09
tetInSphere3000	257	547	0.03

***TABLE 2**: The number of bits that are needed to encode the first sheet are shown. From left to right, the columns contain: the name of the model, the number of back faces, the number of bits needed to encode the connectivity of the back faces with Edgebreaker, the number of these bits per tetrahedron.*

The rows reference meshes shown in Fig. 3.



tet100 tet120 tet2286 tet3000

FIGURE 3: *The various meshes used to test TetStremer (wire-frame and boundary).*

Footprint

To estimate the footprint needed for the decoder, consider a tet mesh with N_T tets and N_V vertices. There is no fixed relation between N_T and N_V, but for large meshes $N_T = r\ N_V$, with $r \in [4.5, 6.5]$, depending on the data and the triangulation algorithm used to generate the tet mesh. Although the number N_E of external faces and vertices (as a function of N_T and r) depends on the data set, we can estimate it for sufficiently large and regular meshes. Assume that the mesh is a sphere of radius R. Its volume is $4\pi R^3/3$ and its surface area is $4\pi R^2$. Assume that it is tetrahedralized with reasonably regular tets, whose faces have an average area A and whose volume on average is V, with $A^3/V^2=k$, with $k=3\sqrt{3}/32$. We have $4\pi R^2=AN_E$ and $4\pi R^3/3=VN_T$, which yields $(AN_E)^3/(VN_T)^2=36\pi$. Hence, $N_E^3=36\pi N_T^2(V^2/A^3)$ and thus $N_E^3=128\pi\sqrt{3}N_T^2$. For example, a mesh with 10K tets will have about 4K external faces, roughly half of which will be front facing. A tet mesh with 100K tets will have less than 20K external tris.

To store a tri mesh of at most T tris, the client needs to store T/2 vertices (each occupying less than 64 bits) and $6T\log_2(T)$ bits to store the connectivity. For example, when T is 10K, the server will need less than 12 bytes per tri, which is 2.4 bytes per tet to store the connectivity information for decoding and processing the tets in the back-to-front.

6. Conclusions

We have proposed an approach for the compression and back-to-front streaming of tet meshes. The connectivity is compressed losslessly down to about 1.7 bits per tet. The client needs only store the evolving sheet, which has a small footprint compared to the tet mesh. The compression and decompression algorithms are simple and fast.

7. Acknowledgements

This research was supported by the DARPA/NSF CARGO grant number 0138420.

8. Bibliography

[1] N. Max, "Optical Methods for Direct Volume Rendering," *IEEE Trans. On Visualization and Computer Graphics*, vol. 1, pp. 99-108, 1995.

[2] M. Deering, "Geometry Compression," presented at 22nd Annual ACM Conference on Computer Graphics, 1995.

[3] M. Isenburg and P. Lindstrom, "Streaming Meshes," LLNL technical report UCRL-CONF-201992, April 2004.

[4] Z. Karni, A. Bogomjakov, and C. Gotsman, "Efficient compression and rendering of multi-resolution meshes," presented at Visualization, 2002.

[5] H. Berk, C. Aykanat, and U. Güdükbay, "Direct volume rendering of unstructured grids," *Computer & Graphics*, vol. 27, pp. 387-406, 2003.

[6] J. Wilhelms and A. V. Gelder, "A Coherent Projection Approach for Direct Volume Rendering," *Computer Graphics*, vol. 25, pp. 275-284, 1991.

[7] N. Max and P. Williams, "Cell Projection of Meshes with Non-Planar Faces," presented at Dagstuhl, 2000.

[8] P. L. Williams, "Visibility-ordering meshed polyhedra," *ACM Transactions on Graphics (TOG)*, vol. 11, 1992.

[9] N. Max, P. Hanrahan, and R. Crawfis, "Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions," presented at Computer Graphics (San Diego Workshop on Volume Visualization), San Diego, 1990.

[10] A. Szymczak and J. Rossignac, "Grow & Fold: Compression of Tetrahedral Meshes," presented at ACM Symposium on Solid Modeling and Applications, Ann Arbor, 1999.

[11] S. Gumhold, S. Guthe, and W. Strasser, "Tetrahedral Mesh Compression with the Cut-Border Machine," presented at IEEE Visualization, San Francisco, 1999.

[12] R. Pajarola, J. Rossignac, and A. Szymczak, "Implant Sprays: Compression of Progressive Tetrahedral Mesh Connectivity," presented at IEEE Visualization, San Francisco, 1999.

[13] C.-k. Yang, T. Mitra, and T.-c. Chiueh, "On-the-fly Rendering Of Losslessly Compressed Irregular Volume Data," presented at VisSym '01, Joint Eurographics - IEEE TCVG Symposium on Visualization, Ascona, 2001.

[14] J. Rossignac, A. Safonova, and A. Szymczak, "Edgebreaker on a Corner Table: A Simple Technique for Representing and Compressing Triangulated Surfaces," presented at Shape Modeling International Conference, Genoa, 2001.

[15] R. Pajarola and J. Rossignac, "Compressed Progressive Meshes," *IEEE Trans. On Visualization and Computer Graphics*, vol. 6, pp. 79-93, 2000.

[16] V. Coors and J. Rossignac, "Guess Connectivity: Delphi Encoding in Edgebreaker," Atlanta 2002.

[17] A. Szymczak and J. Rossignac, "Grow&Fold: Compressing the connectivity of tetrahedral meshes," *Computer-Aided Design*, vol. 32, pp. 527-538, 2000.

[18] *H. Edelsbrunner, "An Acyclicity Theorem for Cell Complexes in d Dimension" Combinatorica, Vol. 10, no. 3, pp. 251-260, 1990.*